Chapter 3

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in Subsection 1.1.2. We investigate three research questions. In the first question, we artificially generate WebAssembly program variants and quantitatively compare the static differences between variants. Our second research question focuses on comparing their behavior during their execution. The final research question evaluates the feasibility of using the program variants in security-sensitive environments such as Edge-Cloud computing proposing a multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use an empirical method [**?**], proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

3.1 Corpora

Our experiments assess the impact of artificially created diversity in terms of program variants size, static and dynamic differences. The first step is to build a suitable corpus of programs' seeds to generate the variants. Finally, we answer all our research questions with three corpora of diverse and representative programs for our experiments. We build our three corpora in an escalating strategy. The first corpus is diverse and contains simple programs in terms of code size, making them easy to analyze manually. The latter two corpora contain more extensive real-world programs, including one project meant for security-sensitive applications. Finally,

all corpora are considered to come along the LLVM pipeline. We base this decision on the previous experimental work of Hilbig et al. [**?**]. This work shows that approximately 65% of all WebAssembly programs come out of C/C++ source code, and more than 75% if Rust is included. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** : We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library ported to WebAssembly in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation. We select 5 programs that interconnect the 102 modules of the project.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Besides, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 3.1 we listed the corpus name, the number of modules, the number of programs inside the corpus, the total number of functions, the range of lines of code, and the original location of the corpus.

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

Corpus	No. modules	No. programs	No. functions	LOC range	Location
Rosetta	-	303	303	7 - 150	`http://rosettacode.org/wiki/Rosetta_Code`
Libsodium	102	5	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total		310	3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of modules, the number of programs, the number of functions, the lines of code range and the location of the corpus.

3.2 RQ1. To what extent can we generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original program, written in C/C++ in the case of the Rosetta corpus and LLVM bitcodes in the case of Libsodium and QrCode. In Figure 3.1 we simplify the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW. To answer RQ1, we study the outcome of this pipeline, the generated variants.

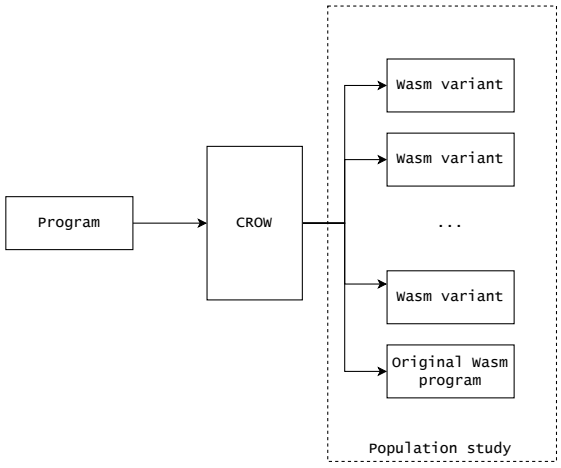

Figure 3.1: The program variants generation for RQ1.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of unique variants for each original function of each corpus.

Metric 1 *Population size $S(P)$: Given a program P and its generated variants V, the population size metric is defined as.*

$$S(P) = |V|$$

Notice that, the variant population includes P as an instance.

A program and its variants compose what we call a program's population. Notice that all proposed metrics over programs and their variants make sense only at the population level. Therefore, we compare semantically equivalent programs from the same population.

Protocol

To generate program variants, we mainly synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [**?**] against the original program. An enumerative synthesis is a brute-force approach to generate program variants. With a maximum number of instructions, it constructs and checks all possible programs up to that limit. For a simplified instance, with a maximum code size of 2 instructions in a programming language with L possible constructions, an enumerative synthesizer builds all $L \times L$ combinations finding program variants. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in our synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and Libsodium is motivated by the properties listed in Table 3.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the $303 + 869 + 1849$ functions in the corpora to CROW, as Figure 3.1 illustrates, to synthesize program variants. Finally, we calculate Metric 1 for each program's population and conclude by answering RQ1.

3.3 RQ2. To what extent are the generated variants dynamically different?

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate the question into two fields as Figure 3.2 illustrates: static comparison and dynamic comparison. The static analysis focuses on the appreciated differences between the program variants between them and against the original program, and we address it in answering RQ1. With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic complements RQ1 and it is essential to provide a full understanding of diversification [?] . We use the original functions from the Rosetta corpus described in Section 3.1 and their variants generated in the answering of RQ1. We use only the Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it.

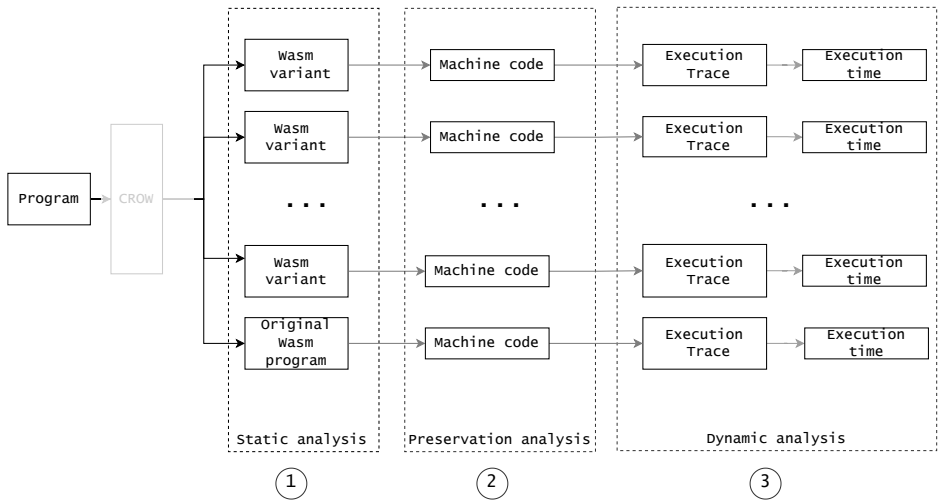

Figure 3.2: Population study methodology for each original corpora and the variants generated in RQ1.

To dynamically compare programs and their variants, we execute each program on each population of variants to collect their execution traces and execution times. We perform fine-grained comparisons by studying all pairs of programs in the population of variants plus the original program. Therefore, the defined metrics are formulated to support a pairwise comparison strategy. In the following, we define

the metrics used to answer RQ2.

Metrics

We measure the difference between programs at runtime by comparing their execution times and execution traces. We compare their execution traces with an alignment metric at the function and instruction level. We propose a global alignment approach using Dynamic Time Warping (DTW) for their execution traces. Dynamic Time Warping [**?**] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. In the following, we define the *dt_dyn* metric.

Metric 2 *dt_dyn: Given two programs P and P' from the same program's population, dt_dyn(P,P'), computes the DTW distance between the traces collected during their execution.*
A dt_dyn of 0 *means that both traces are identical. The higher the value, the more different the traces.*

In our experiments, a stack operation trace is the consecutive list of `push` and `pop` operations performed by the WebAssembly engine during the execution of the program.

We use the execution time of the programs in the population to complement the answer to RQ2. We compare each execution time distribution from the variants against the distribution of the original program.

Metric 3 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated in the answering of RQ1 in a pairwise comparison using Metric 2. We use SWAM[2] to execute each program and variant to collect the stack operation traces in the case of the Rosetta corpus. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations. We want to remark that we only collect the stack operation traces due to the memory-agnosticism of our approach to generate variants. Our approach does not change the memory-like operations of the original code.

Furthermore, we collect the execution time, Metric 3, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [**?**] in a pairwise strategy.

[2] `https://github.com/satabin/swam`

3.4 RQ3. To what extent the artificial variants exhibit different execution times on Edge-Cloud platforms?

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant execution binaries. For this purpose, we build multivariant binaries to be deployed at the Edge. We use the variants generated for the programs of the Libsodium and QrCode corpora, $2+5$ programs involving $869+1849$ functions respectively. For this research question, 7 Multivariant binaries are created by converting each program's population of variants into a single function for which each call at runtime selects and executes a different variant. We illustrate the protocol to answer RQ3 in Figure 3.3.

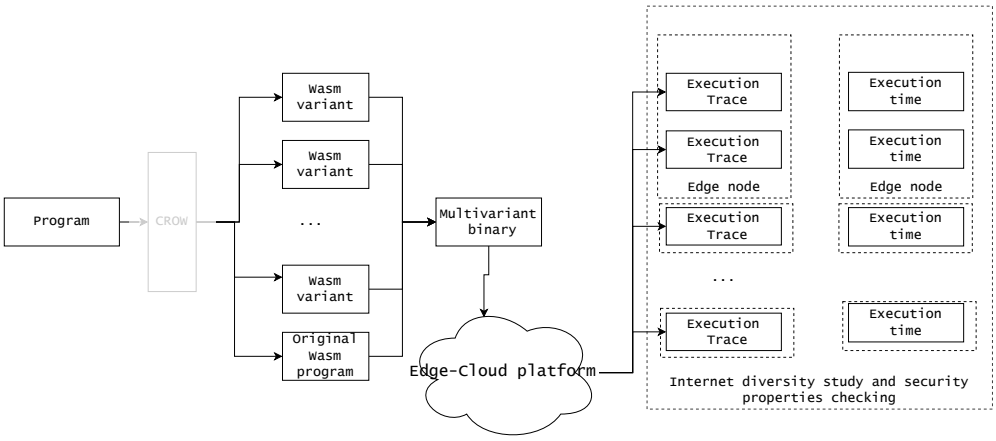

Figure 3.3: Multivariant binary creation and workflow for RQ3.

We assess the ability of multivariant binaries to exhibit random execution paths when executed on edge. We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes to assess Multivariant Execution (MVE) worldwide. Finally, we measure the differences for the execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

Metrics

To compare the diversity of function traces, we enunciate the following metrics.

Metric 4 *Unique traces: $R(n,e)$. Let $S(n,e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one program e on an edge node n, $H(n,e)$ the collection*

of hashes of each trace and $U(n, e)$ the set of unique trace hashes in $H(n, e)$. The uniqueness ratio of traces collected for edge node n and program e is defined as

$$R(n, e) = \frac{|U(n, e)|}{|H(n, e)|}$$

Metric 5 *Normalized Shannon entropy: $E(e)$ Let e be a program, $C(e) = \cdot_{n=0}^{64} H(n, e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the program e over the collected traces is defined as:*

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy, Metric 5, by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between programs. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the program consistently exhibits the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, each edge node executes a different path every time the program is requested.

Protocol

We run the experiments to answer RQ3 on the Edge. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. These edge nodes usually have an arbitrary and heterogeneous composition in architecture and CPU models.

We execute each program 100 times on each node to measure the diversity of execution traces exhibited by the multivariant binaries. Each query on the same program is performed with the same input value. This guarantees that if we observe different traces for different executions, it is due to the presence of multiple function variants.

For each query, we collect the execution trace, i.e., the sequence of function names that have been executed when triggering the query. We instrument the multivariant binaries to record each function entrance to observe these traces.

We measure the number of unique execution traces exhibited by each multivariant binary, Metric 4, on each separate edge node. To compare the traces, we hash them with the `md5sum` function. We calculate the number of unique hashes among the 100 traces collected for a program on one edge node. We follow by collecting the normalized Shannon entropy, Metric 5, for all collected execution traces for each program. The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency of appearing in part of

the sampling, then it is inevitable that this trace will appear in the other part of the sampling.

We calculate Metric 5 for the seven programs, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per program. Each trace is collected in a round-robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before collecting the second trace. This process is followed until 100 traces are collected from all edge nodes.

In addition, we collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [**?**] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

3.5 Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. Finally, we propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Nevertheless, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.